Memory, Data, & Addressing II
CSE 351 Spring 2019

Instructor:
Ruth Anderson

Teaching Assistants:
Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh

http://xkcd.com/138/
Administrivia

- Lab 0 due Monday @ 11:59 pm
  - You will be revisiting this program throughout this class!

- Homework 1 due Wednesday
  - Reminder: autograded, 20 tries, no late submissions

- Lab 1a released Monday, due next Monday (4/15)
  - Pointers in C
  - Reminder: last submission graded, individual work
Lab Late Days

- All submissions due at 11:59 pm
  - Count lateness in *days* (**even if just by a second**)
    - Special: weekends count as *one day*
  - No submissions accepted more than two “late days” late

- You are given **5 lab late days** for the whole quarter
  - Late days only apply to Labs (not HW)
  - No benefit to having leftover late days
  - If you use more than 5 late days, late penalty is 20% deduction per day
  - Penalties applied at end of quarter to *maximize* your grade

- Use at own risk – don’t want to fall too far behind
  - Intended to allow for unexpected circumstances
Review Questions

1) If the word size of a machine is 64-bits, which of the following is usually true? (pick all that apply)
   a) 64 bits is the size of a pointer  
      
   b) 64 bits is the size of an integer  
      
   c) 64 bits is the width of a register

2) (True/False) By looking at the bits stored in memory, I can tell if a particular 4-bytes is being used to represent an integer, floating point number, or instruction.

3) If the size of a pointer on a machine is 6 bits, the address space is how many bytes?
   \[ 2^6 \]
Memory, Data, and Addressing

- Hardware - High Level Overview
- Representing information as bits and bytes
  - Memory is a byte-addressable array
  - Machine “word” size = address size = register size
- Organizing and addressing data in memory
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations
Addresses and Pointers in C

- `&` = “address of” operator
- `*` = “value at address” or “dereference” operator

Declares a variable, `ptr`, that is a pointer to (i.e. holds the address of) an `int` in memory

Declares two variables, `x` and `y`, that hold `ints`, and initializes them to 5 and 2, respectively

Sets `ptr` to the address of `x` (“`ptr` points to `x`”)

Sets `y` to “1 plus the value stored at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y=1+x`;

What is `*(&y)`?

* is also used with variable declarations

Prefix `int*` declares a variable that is a pointer to `int`

Postfix `int` declares a variable that holds an `int`
Assignment in C

- A variable is represented by a location in memory
- Declaration ≠ initialization (initially holds “garbage”)
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18
Assignment in C

- A variable is represented by a location in memory
- Declaration ≠ initialization (initially holds “garbage”)
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18

32-bit example (pointers are 32-bits wide)

little-endian
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- `int x, y;`
- `x = 0;`

32-bit example (pointers are 32-bits wide)

\& = “address of”
* = “dereference”
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- `int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, store in x
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- `int x, y;
  x = 0;
  y = 0x3CD02700;
  x = y + 3;
  Get value at y, add 3, store in x

- `int* z;
  z is at address 0x20`
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- `int x, y;
  x = 0;
  y = 0x3CD02700;
  x = y + 3;
  - Get value at y, add 3, store in x

- `int* z = &y + 3;`  // expect 0x1b
  - Get address of y, "add 3", store in z

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”

Pointer arithmetic
Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, `sizeof(int) = 4`

```c
int* z = &y + 3;
```
- Get address of `y`, add `3*sizeof(int)`, store in `z`
  - `&y = 0x18 = 1*16^1 + 8*16^0 = 24`
  - `24 + 3*(4) = 36 = 2*16^1 + 4*16^0 = 0x24`

- Pointer arithmetic can be dangerous!
  - Can easily lead to bad memory accesses
  - Be careful with data types and casting
Assignment in C

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int* z = &y + 3;`
  - Get address of `y`, add 12, store in `z`
- `*z = y;`
  - What does this do?

---

32-bit example (pointers are 32-bits wide)

`&` = “address of”

`*` = “dereference”
Assignment in C

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int* z = &y + 3;`
  - Get address of `y`, add 12, store in `z`
- `*z = y;`
  - Get value of `y`, put in address stored in `z`

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”

The target of a pointer is also a location
Assignment in C - Handout

- left-hand side = right-hand side;
  - LHS must evaluate to a *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`

- `int* z = &y + 3;`
  - Get address of `y`, add 12, store in `z`

- `*z = y;`
  - Get value of `y`, put in address stored in `z`

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”
Arrays in C

Declaration: `int a[6];`  // `&a` is `0x10`

Arrays are adjacent locations in memory storing the same type of data object.

`a` (array name) returns the array’s address.

64-bit example (pointers are 64-bits wide)
Arrays in C

Declaration: `int a[6];`

Indexing: `a[0] = 0x015f;`
`a[5] = a[0];`

Arrays are adjacent locations in memory storing the same type of data object.

- `a` (array name) returns the array’s address
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes
Arrays in C

Declaration: `int a[6];`

Indexing:  
- `a[0] = 0x015f;`
- `a[5] = a[0];`

No bounds  
- `a[6] = 0xBAD;`
checking:  
- `a[-1] = 0xBAD;`

Arrays are adjacent locations in memory storing the same type of data object.

- `a` (array name) returns the array’s address.
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes.

Arrays are stored in memory as follows:

```
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7
0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
```

- `a[0]` is stored at `0x00` and contains `0x015f`.
- `a[2]` is stored at `0x08` and contains `0x0100`.
- `a[4]` is stored at `0x10` and contains `0x0100`.

```
&a[0]  5F 01 00 00
&a[2]  5F 01 00 00
&a[4]  5F 01 00 00
&a[6]  AD OB 00 00
```

```
0x00
0x08
0x10
0x18
0x20
0x28
0x30
0x38
0x40
0x48
```
Arrays in C

Declaration: `int a[6];`

Indexing: 
- `a[0] = 0x015f;`
- `a[5] = a[0];`

No bounds checking: 
- `a[6] = 0xBAD;`
- `a[-1] = 0xBAD;`

Pointers: 
- `int* p;`
- `p = a;`
- `p = &a[0];`
- `*p = 0xA;`

Arrays are adjacent locations in memory storing the same type of data object.

- `a` (array name) returns the array’s address.
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes.
Arrays in C

Declaration: `int a[6];`

Indexing:  
- `a[0] = 0x015f;`
- `a[5] = a[0];`

No bounds: `a[6] = 0xBAD;`

(checking: `a[-1] = 0xBAD;`)

Pointers:  
- `int* p;`
- `p = a;`
- `p = &a[0];`
- `*p = 0xA;`
- `p[1] = 0xB;`
- `*(p+1) = 0xB;`
- `p = p + 2;`

Array indexing = address arithmetic (both scaled by the size of the type)

Equivalent:  
- `p[1] = 0xB;`
- `*(p+1) = 0xB;`
- `0x10 + 1 \rightarrow 0x14`
- `p = p + 2;`

Pointer arithmetic:  
- `0x10 + 2 \rightarrow 0x18`

Arrays are adjacent locations in memory storing the same type of data object

- `a` (array name) returns the array’s address
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes

$ p[i] \iff *(p+i) $
Arrays in C

Declaration: `int a[6];`

Indexing: `a[0] = 0x015f;`  
`a[5] = a[0];`

No bounds checking: `a[6] = 0xBAD;`

Pointers: `int* p;`  
`p = a;`  
`p = &a[0];`  
`*p = 0xA;`

Array indexing = address arithmetic (both scaled by the size of the type)

`p[1] = 0xB;`  
`*(p+1) = 0xB;`  
`p = p + 2;`

`*p = a[1] + 1;`

Arrays are adjacent locations in memory storing the same type of data object.

`a` (array name) returns the array’s address.

`&a[i]` is the address of `a[0]` plus `i` times the element size in bytes.
Arrays in C - Handout

Declaration: \texttt{int a[6];}

Indexing: \texttt{a[0] = 0x015f;}
\texttt{a[5] = a[0];}

No bounds \texttt{a[6] = 0xBAD;}
checking: \texttt{a[-1] = 0xBAD;}

Pointers: \texttt{int* p;}
equivalent \begin{cases} 
\texttt{p = a;} \\
\texttt{p = \&a[0];} \\
\texttt{*p = 0xA;}
\end{cases}

Arrays are adjacent locations in memory storing the same type of data object
\texttt{a} (array name) returns the array’s address
\texttt{\&a[i]} is the address of \texttt{a[0]} plus \texttt{i} times the element size in bytes

array indexing = address arithmetic (both scaled by the size of the type)
equivalent \begin{cases} 
\texttt{p[1] = 0xB;} \\
\texttt{*(p+1) = 0xB;} \\
\texttt{p = p + 2;}
\end{cases}
\texttt{*p = a[1] + 1;}
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?

- Vote at http://PollEv.com/rea

```c
void main() {
    int a[] = {5,10};
    int* p = a;
    p = p + 1; // sizeof(int) = 4
    *p = *p + 1;
}
```

<table>
<thead>
<tr>
<th>Address (decimal)</th>
<th>Data (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>a[0] 5</td>
</tr>
<tr>
<td></td>
<td>a[1] 10</td>
</tr>
<tr>
<td></td>
<td>p 104</td>
</tr>
</tbody>
</table>

- (A) 101 10 5 then 101 11 5 11
- (B) 104 10 5 then 104 11 5 11
- (C) 100 6 6 then 101 6 6 10
- (D) 100 6 6 then 104 6 6 10
Representing strings

- C-style string stored as an array of bytes (**char**
  
  - Elements are one-byte ASCII codes for each character
  - No “String” keyword, unlike Java

<table>
<thead>
<tr>
<th>32</th>
<th>space</th>
<th>48</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>39</td>
<td>’</td>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>58</td>
<td>:</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>59</td>
<td>;</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>Q</td>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>~</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ASCII:** American Standard Code for Information Interchange
Null-Terminated Strings

- **Example:** "Donald Trump" stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal</th>
<th>68</th>
<th>111</th>
<th>110</th>
<th>97</th>
<th>108</th>
<th>100</th>
<th>32</th>
<th>84</th>
<th>114</th>
<th>117</th>
<th>109</th>
<th>112</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>0x44</td>
<td>0x6F</td>
<td>0x6E</td>
<td>0x61</td>
<td>0x6C</td>
<td>0x64</td>
<td>0x20</td>
<td>0x54</td>
<td>0x72</td>
<td>0x75</td>
<td>0x6D</td>
<td>0x70</td>
<td>0x00</td>
</tr>
<tr>
<td>Text</td>
<td>D o n a l d T r u m p</td>
<td>&quot;\0&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Last character followed by a 0 byte (\' \0 ') (a.k.a. "null terminator")
  - Must take into account when allocating space in memory
  - Note that \' 0 \' \(\neq\) \' \0 ' (i.e. character 0 has non-zero value)

- **How do we compute the length of a string?**
  - Traverse array until null terminator encountered
Endianness and Strings

```c
char s[6] = "12345";
```

- **Byte ordering (endianness) is not an issue for 1-byte values**
  - The whole array does not constitute a single value
  - Individual elements are values; chars are single bytes
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to char
  - C has unchecked casts  !! DANGER !!

```c
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p	0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

**printf directives:**
- `%p`  Print pointer
- `	`  Tab
- `%x`  Print value as hex
- `
`  New line
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to char
  - C has unchecked casts!! DANGER!!

```c
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");

    // Pointer arithmetic on char*
}
```

```c
void show_int(int x) {
    show_bytes((char*) &x, sizeof(int));
}
```
show_bytes Execution Example

```c
int x = 12345; // 0x00003039
printf("int x = %d;\n", x);
show_int(x); // show_bytes((char *) &x, sizeof(int));
```

- Result (Linux x86-64):
  - **Note**: The addresses will change on each run (try it!), but fall in same general range

```c
int x = 12345;
0x7fffb7f71dbc 0x39
0x7fffb7f71dbd 0x30
0x7fffb7f71dbe 0x00
0x7fffb7f71dbf 0x00
```
Summary

- Assignment in C results in value being put in memory location
- Pointer is a C representation of a data address
  - `&` = “address of” operator
  - `*` = “value at address” or “dereference” operator
- Pointer arithmetic scales by size of target type
  - Convenient when accessing array-like structures in memory
  - Be careful when using particularly when casting variables
- Arrays are adjacent locations in memory storing the same type of data object
  - Strings are null-terminated arrays of characters (ASCII)